# R 54

## 54.1 RCL/RCR/ROL/ROR-—Rotate

| Opcode | Instruction | Description |
|---|---|---|
| D0 /2 | RCL *r/m8*,1 | Rotate 9 bits (CF,*r/m8*) left once |
| D2 /2 | RCL *r/m8*,CL | Rotate 9 bits (CF,*r/m8*) left CL times |
| C0 /2 *ib* | RCL *r/m8,imm8* | Rotate 9 bits (CF,*r/m8*) left *imm8* times |
| D1 /2 | RCL *r/m16*,1 | Rotate 17 bits (CF,*r/m16*) left once |
| D3 /2 | RCL *r/m16*,CL | Rotate 17 bits (CF,*r/m16*) left CL times |
| C1 /2 *ib* | RCL *r/m16,imm8* | Rotate 17 bits (CF,*r/m16*) left *imm8* times |
| D1 /2 | RCL *r/m32*,1 | Rotate 33 bits (CF,*r/m32*) left once |
| D3 /2 | RCL *r/m32*,CL | Rotate 33 bits (CF,*r/m32*) left CL times |
| C1 /2 *ib* | RCL *r/m32,imm8* | Rotate 33 bits (CF,*r/m32*) left *imm8* times |
| D0 /3 | RCR *r/m8*,1 | Rotate 9 bits (CF,*r/m8*) right once |
| D2 /3 | RCR *r/m8*,CL | Rotate 9 bits (CF,*r/m8*) right CL times |
| C0 /3 *ib* | RCR *r/m8,imm8* | Rotate 9 bits (CF,*r/m8*) right *imm8* times |
| D1 /3 | RCR *r/m16*,1 | Rotate 17 bits (CF,*r/m16*) right once |
| D3 /3 | RCR *r/m16*,CL | Rotate 17 bits (CF,*r/m16*) right CL times |
| C1 /3 *ib* | RCR *r/m16,imm8* | Rotate 17 bits (CF,*r/m16*) right *imm8* times |
| D1 /3 | RCR *r/m32*,1 | Rotate 33 bits (CF,*r/m32*) right once |
| D3 /3 | RCR *r/m32*,CL | Rotate 33 bits (CF,*r/m32*) right CL times |
| C1 /3 *ib* | RCR *r/m32,imm8* | Rotate 33 bits (CF,*r/m32*) right *imm8* times |
| D0 /0 | ROL *r/m8*,1 | Rotate 8 bits *r/m8* left once |
| D2 /0 | ROL *r/m8*,CL | Rotate 8 bits *r/m8* left CL times |
| C0 /0 *ib* | ROL *r/m8,imm8* | Rotate 8 bits *r/m8* left *imm8* times |
| D1 /0 | ROL *r/m16*,1 | Rotate 16 bits *r/m16* left once |
| D3 /0 | ROL *r/m16*,CL | Rotate 16 bits *r/m16* left CL times |
| C1 /0 *ib* | ROL *r/m16,imm8* | Rotate 16 bits *r/m16* left *imm8* times |
| D1 /0 | ROL *r/m32*,1 | Rotate 32 bits *r/m32* left once |
| D3 /0 | ROL *r/m32*,CL | Rotate 32 bits *r/m32* left CL times |
| C1 /0 *ib* | ROL *r/m32,imm8* | Rotate 32 bits *r/m32* left *imm8* times |
| D0 /1 | ROR *r/m8*,1 | Rotate 8 bits *r/m8* right once |
| D2 /1 | ROR *r/m8*,CL | Rotate 8 bits *r/m8* right CL times |
| C0 /1 *ib* | ROR *r/m8,imm8* | Rotate 8 bits *r/m16* right *imm8* times |
| D1 /1 | ROR *r/m16*,1 | Rotate 16 bits *r/m16* right once |
| D3 /1 | ROR *r/m16*,CL | Rotate 16 bits *r/m16* right CL times |
| C1 /1 *ib* | ROR *r/m16,imm8* | Rotate 16 bits *r/m16* right *imm8* times |
| D1 /1 | ROR *r/m32*,1 | Rotate 32 bits *r/m32* right once |
| D3 /1 | ROR *r/m32*,CL | Rotate 32 bits *r/m32* right CL times |
| C1 /1 *ib* | ROR *r/m32,imm8* | Rotate 32 bits *r/m32* right *imm8* times |

## Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location (see “ROL, ROR, RCL, and RCR Instruction Operations”). The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location (see “ROL, ROR, RCL, and RCR Instruction Operations”).

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag (see “ROL, ROR, RCL, and RCR Instruction Operations”). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see “ROL, ROR, RCL, and RCR Instruction Operations”). For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

## Intel Architecture Compatibility

The 8086 does not mask the rotation count. However, all other Intel Architecture processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

## Operation

```
(* RCL and RCR instructions *)
SIZE ← OperandSize
CASE (determine count) OF
    SIZE = 8:    tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE = 16:   tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE = 32:   tempCOUNT ← COUNT AND 1FH;
ESAC;
(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST * 2) + CF;
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
(* RCR instruction operation *)
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
```

```
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (CF * 2^SIZE);
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
(* ROL and ROR instructions *)
SIZE ← OperandSize
CASE (determine count) OF
    SIZE = 8:    tempCOUNT ← COUNT MOD 8;
    SIZE = 16:   tempCOUNT ← COUNT MOD 16;
    SIZE = 32:   tempCOUNT ← COUNT MOD 32;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST * 2) + tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
CF ← LSB(DEST);
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
(* ROR instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (tempCF * 2^SIZE);
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
CF ← MSB(DEST);
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
    ELSE OF is undefined;
FI;
```

## Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for single-bit rotates (see “Description” above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the source operand is located in a nonwritable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a null segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

## 54.2 RDMSR—Read from Model Specific Register

| Opcode | Instruction | Description |
|---|---|---|
| 0F 32 | RDMSR | Load MSR specified by ECX into EDX:EAX |

### Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. If less than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, *Model-Specific Registers (MSRs)*, in the *Intel Architecture Software Developer's Manual, Volume 3*, lists all the MSRs that can be read with this instruction and their addresses.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

### Intel Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the Intel Architecture with the Pentium processor. Execution of this instruction by an Intel Architecture processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

```
EDX:EAX ← MSR[ECX];
```

### Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If the value in ECX specifies a reserved or unimplemented MSR address. |

### Real-Address Mode Exceptions

#GP If the value in ECX specifies a reserved or unimplemented MSR address.

### Virtual-8086 Mode Exceptions

#GP(0) The RDMSR instruction is not recognized in virtual-8086 mode.

## 54.3 RDPMC—Read Performance-Monitoring Counters

| Opcode | Instruction | Description |
|---|---|---|
| 0F 33 | RDPMC | Read performance-monitoring counter specified by ECX into EDX:EAX |

### Description

Loads the contents of the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. The Pentium Pro processor has two performance-monitoring counters (0 and 1), which are specified by placing 0000H or 0001H, respectively, in the ECX register.

The RDPMC instruction allows application code running at a privilege level of 1, 2, or 3 to read the performance-monitoring counters if the PCE flag in the CR4 register is set. This instruction is provided to allow performance monitoring by application code without incurring the overhead of a call to an operating-system procedure.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, *Performance Monitoring Counters*, in the *Intel Architecture Software Developer's Manual, Volume 3*, lists all the events that can be counted.

The RDPMC instruction does not serialize instruction execution. That is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must use a serializing instruction (such as the CPUID instruction) before and/or after the execution of the RDPCM instruction.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to determine the counter to access and a full 40-bit result is returned (the low-order 32 bits in the EAX register and the high-order 9 bits in the EDX register).

### Intel Architecture Compatibility

The RDPMC instruction was introduced into the Intel Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The other Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

### Operation

```
IF (ECX = 0 OR 1) AND ((CR4.PCE = 1) OR ((CR4.PCE = 0) AND (CPL=0)))
    THEN
        EDX:EAX ← PMC[ECX];
    ELSE (* ECX is not 0 or 1 and/or CR4.PCE is 0 and CPL is 1, 2, or 3 *)
```

```
        #GP(0); FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.

If the value in the ECX register is not 0 or 1.

### Real-Address Mode Exceptions

#GP If the PCE flag in the CR4 register is clear.

If the value in the ECX register is not 0 or 1.

### Virtual-8086 Mode Exceptions

#GP(0) If the PCE flag in the CR4 register is clear.

If the value in the ECX register is not 0 or 1.

## 54.4 RDTSC—Read Time-Stamp Counter

| Opcode | Instruction | Description |
|---|---|---|
| 0F 31 | RDTSC | Read time-stamp counter into EDX:EAX |

### Description

Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.

This instruction was introduced into the Intel Architecture in the Pentium processor.

### Operation

```
IF (CR4.TSD = 0) OR ((CR4.TSD = 1) AND (CPL=0))
    THEN
        EDX:EAX ← TimeStampCounter;
    ELSE (* CR4 is 1 and CPL is 1, 2, or 3 *)
        #GP(0)
```

```
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.

### Real-Address Mode Exceptions

#GP If the TSD flag in register CR4 is set.

### Virtual-8086 Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set.

## 54.5 REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

| Opcode | Instruction | Description |
|---|---|---|
| F3 6C | REP INS *r/m8*, DX | Input (E)CX bytes from port DX into ES:[(E)DI] |
| F3 6D | REP INS *r/m16*,DX | Input (E)CX words from port DX into ES:[(E)DI] |
| F3 6D | REP INS *r/m32*,DX | Input (E)CX doublewords from port DX into ES:[(E)DI] |
| F3 A4 | REP MOVS *m8,m8* | Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI] |
| F3 A5 | REP MOVS *m16,m16* | Move (E)CX words from DS:[(E)SI] to ES:[(E)DI] |
| F3 A5 | REP MOVS *m32,m32* | Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI] |
| F3 6E | REP OUTS DX,*r/m8* | Output (E)CX bytes from DS:[(E)SI] to port DX |
| F3 6F | REP OUTS DX,*r/m16* | Output (E)CX words from DS:[(E)SI] to port DX |
| F3 6F | REP OUTS DX,*r/m32* | Output (E)CX doublewords from DS:[(E)SI] to port DX |
| F3 AC | REP LODS AL | Load (E)CX bytes from DS:[(E)SI] to AL |
| F3 AD | REP LODS AX | Load (E)CX words from DS:[(E)SI] to AX |
| F3 AD | REP LODS EAX | Load (E)CX doublewords from DS:[(E)SI] to EAX |
| F3 AA | REP STOS *m8* | Fill (E)CX bytes at ES:[(E)DI] with AL |
| F3 AB | REP STOS *m16* | Fill (E)CX words at ES:[(E)DI] with AX |
| F3 AB | REP STOS *m32* | Fill (E)CX doublewords at ES:[(E)DI] with EAX |
| F3 A6 | REPE CMPS *m8,m8* | Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI] |
| F3 A7 | REPE CMPS *m16,m16* | Find nonmatching words in ES:[(E)DI] and DS:[(E)SI] |
| F3 A7 | REPE CMPS *m32,m32* | Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI] |
| F3 AE | REPE SCAS *m8* | Find non-AL byte starting at ES:[(E)DI] |
| F3 AF | REPE SCAS *m16* | Find non-AX word starting at ES:[(E)DI] |
| F3 AF | REPE SCAS *m32* | Find non-EAX doubleword starting at ES:[(E)DI] |
| F2 A6 | REPNE CMPS *m8,m8* | Find matching bytes in ES:[(E)DI] and DS:[(E)SI] |
| F2 A7 | REPNE CMPS *m16,m16* | Find matching words in ES:[(E)DI] and DS:[(E)SI] |
| F2 A7 | REPNE CMPS *m32,m32* | Find matching doublewords in ES:[(E)DI] and DS:[(E)SI] |
| F2 AE | REPNE SCAS *m8* | Find AL, starting at ES:[(E)DI] |
| F2 AF | REPNE SCAS *m16* | Find AX, starting at ES:[(E)DI] |
| F2 AF | REPNE SCAS *m32* | Find EAX, starting at ES:[(E)DI] |

## Description

Repeats a string instruction the number of times specified in the count register ((E)CX) or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

All of these repeat prefixes cause the associated instruction to be repeated until the count in register (E)CX is decremented to 0 (see the following table). (If the current address-size attribute is 32, register ECX is used as a counter, and if the address-size attribute is 16, the CX register is used.) The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the (E)CX register with a JECXZ instruction or by testing the ZF flag with a JZ, JNZ, and JNE instruction.

| Repeat Prefix | Termination Condition 1 | Termination Condition 2 |
|---|---|---|
| REP | ECX=0 | None |
| REPE/REPZ | ECX=0 | ZF=0 |
| REPNE/REPNZ | ECX=0 | ZF=1 |

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute.

A REP STOS instruction is the fastest way to initialize a large block of memory.

## Operation

```
IF AddressSize = 16
    THEN
        use CX for CountReg;
    ELSE (* AddressSize = 32 *)
```

```
        use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        service pending interrupts (if any);
        execute associated string instruction;
        CountReg ← CountReg – 1;
        IF CountReg = 0
            THEN exit WHILE loop
        FI;
        IF (repeat prefix is REPZ or REPE) AND (ZF=0)
        OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)
            THEN exit WHILE loop
        FI;
    OD;
```

### Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

### Exceptions (All Operating Modes)

None; however, exceptions can be generated by the instruction a repeat prefix is associated with.

## 54.6 RET—Return from Procedure

| Opcode | Instruction | Description |
|---|---|---|
| C3 | RET | Near return to calling procedure |
| CB | RET | Far return to calling procedure |
| C2 *iw* | RET *imm16* | Near return to calling procedure and pop *imm16* bytes from stack |
| CA *iw* | RET *imm16* | Far return to calling procedure and pop *imm16* bytes from stack |

### Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See “Calling Procedures Using CALL and RET”, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure’s stack and the calling procedure’s stack (that is, the stack being returned to).

## Operation

```
(* Near return *)
IF instruction = near return
    THEN;
        IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                EIP ← Pop();
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack not within stack limits
                    THEN #SS(0)
                FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits THEN #GP(0); FI;
                EIP ← tempEIP;
        FI;
    IF instruction has immediate operand
        THEN IF StackAddressSize=32
            THEN
                ESP ← ESP + SRC; (* release parameters from stack *)
            ELSE (* StackAddressSize=16 *)
                SP ← SP + SRC; (* release parameters from stack *)
        FI;
    FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) OR (PE = 1 AND VM = 1)) AND instruction = far return
    THEN;
        IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack not within stack limits THEN #SS(0); FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits THEN #GP(0); FI;
                EIP ← tempEIP;
                CS ← Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN
            SP ← SP + (SRC AND FFFFH); (* release parameters from stack *)
```

```
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 AND VM = 0) AND instruction = far RET
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits THEN #SS(0); FI;
        FI;
    IF return code segment selector is null THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond diescriptor table limit
        THEN GP(selector; FI;
    Obtain descriptor to which return code segment selector points from descriptor table
    IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
    if return code segment selector RPL < CPL THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
        AND return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
    IF return code segment descriptor is not present THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
    FI;
END;FI;

RETURN-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within ther return code segment limit
        THEN #GP(0);
    FI;
    IF OperandSize=32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
            ESP ← ESP + SRC; (* release parameters from stack *)
        ELSE (* OperandSize=16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
            ESP ← ESP + SRC; (* release parameters from stack *)
    FI;

RETURN-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize=32)
        OR top (8 + SRC) bytes of stack are not within stack limits (OperandSize=16)
            THEN #SS(0); FI;
    FI;
    Read return segment selector;
    IF stack segment selector is null THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
            THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
        OR stack segment is not a writable data segment
        OR stack segment descriptor DPL ≠ RPL of the return code segment selector
            THEN #GP(selector); FI;
        IF stack segment not present THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit THEN #GP(0);
FI:
     CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize=32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
              (* segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            ESP ← ESP + SRC; (* release parameters from called procedure's stack *)
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
              (* segment descriptor information also loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize=16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            ESP ← ESP + SRC; (* release parameters from called procedure's stack *)
            tempESP ← Pop();
            tempSS ← Pop(); (* 16-bit pop; segment descriptor information also loaded *)
              (* segment descriptor information also loaded *)
            ESP ← tempESP;
            SS ← tempSS;
    FI;
```

```
FOR each of segment register (ES, FS, GS, and DS)
     DO;
          IF segment register points to data or non-conforming code segment
          AND CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
               THEN (* segment register invalid *)
                    SegmentSelector ← 0; (* null segment selector *)
          FI;
     OD;
For each of ES, FS, GS, and DS
DO
     IF segment selector index is not within descriptor table limits
          OR segment descriptor indicates the segment is not a data or
               readable code segment
          OR if the segment is a data or non-conforming code segment and the segment
               descriptor's DPL < CPL or RPL of code segment's segment selector
               THEN
                    segment selector register ← null selector;
OD;
ESP ← ESP + SRC; (* release parameters from calling procedure's stack *)
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the return code or stack segment selector null. |
| | If the return instruction pointer is not within the return code segment limit |
| #GP(selector) | If the RPL of the return code segment selector is less then the CPL. |
| | If the return code or stack segment selector index is not within its descriptor table limits. |
| | If the return code segment descriptor does not indicate a code segment. |
| | If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector |
| | If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector |
| | If the stack segment is not a writable data segment. |
| | If the stack segment selector RPL is not equal to the RPL of the return code segment selector. |
| | If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. |
| #SS(0) | If the top bytes of stack are not within stack limits. |
| | If the return stack segment is not present. |
| #NP(selector) | If the return code segment is not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the return instruction pointer is not within the return code segment limit |
| #SS | If the top bytes of stack are not within stack limits. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the return instruction pointer is not within the return code segment limit |
| #SS(0) | If the top bytes of stack are not within stack limits. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If an unaligned memory access occurs when alignment checking is enabled. |

## 54.7 ROL/ROR—Rotate

See entry for RCL/RCR/ROL/ROR—Rotate.

## 54.8 RSM—Resume from System Management Mode

| Opcode | Instruction | Description |
|---|---|---|
| 0F AA | RSM | Resume operation of interrupted program |

### Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SSM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium® and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

See Chapter 11, *System Management Mode (SMM)*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information about SMM and the behavior of the RSM instruction.

### Operation

```
ReturnFromSSM;
ProcessorState ← Restore(SSMDump);
```

### Flags Affected

All.

### Protected Mode Exceptions

| | |
|---|---|
| #UD | If an attempt is made to execute this instruction when the processor is not in SMM. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If an attempt is made to execute this instruction when the processor is not in SMM. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | If an attempt is made to execute this instruction when the processor is not in SMM. |